Portada/ipn_logo.png

Proyecto Final

Acosta Castro Edgar Alberto

Programación II, ESFM-IPN, Ciudad de México

Portada/esfm_logo.png

Buenas tardes, profesor y compañeros. En este video, nuestro equipo —el equipo número 11— hablará sobre la implementación de los árboles en la programación orientada a objetos usando Java.

El equipo está conformado por: Gama Flores Maximiliano, Blanco López César, Josué Enrique García Ayala y su servidor, Acosta Castro Edgar Alberto.

Este video está dividido en tres secciones:

- La primera, correspondiente al programa que hemos nombrado como **Árboles 1**.
- La segunda, titulada **Árboles Recorrido**, que llamamos **Árboles 2**.
- Y la tercera parte, **Árboles con pila**, o **Árboles 3**.

Clase `TreeNode.java`

Comenzamos entonces con el análisis del primer programa.

Estamos utilizando el entorno de desarrollo NetBeans, donde podemos ver el proyecto cargado con las clases relacionadas a los árboles.

Este módulo contiene cuatro clases principales:

- `BinarySearchTree`
- `BinarySearchTreeG`
- `TreeNode`
- `TreePanel`

En esta primera parte nos enfocaremos en la clase `TreeNode`, que es la base de la estructura del árbol.

Esta clase representa cada nodo del árbol binario de búsqueda. Contiene tres elementos clave:

- Una variable entera llamada `value`, donde se almacenará el dato principal del nodo.
- Una referencia a la izquierda llamada `left`.
- Y una referencia a la derecha llamada `right`.

Estas dos últimas referencias (`left` y `right`) nos permiten conectar cada nodo con sus posibles hijos izquierdo y derecho, formando así la estructura jerárquica característica de un árbol binario.

Al construir un nodo, el constructor asigna el valor recibido como argumento a la variable `value`, y además inicializa ambas referencias (`left` y `right`) como `null`.

Esto significa que el nodo, al momento de crearse, no tiene hijos, pero está listo para conectarse en la estructura del árbol.

La clase `TreeNode` es fundamental, ya que será utilizada por las demás clases para construir y visualizar el árbol completo. Comprender cómo funciona esta clase es clave para entender el resto del programa.

Clase `BinarySearchTreeG.java`

Ahora continuamos con la siguiente clase, que se encarga de organizar y construir el árbol binario de búsqueda. Esta clase se llama `BinarySearchTreeG`.

Lo primero que observamos es que esta clase contiene un único atributo privado llamado `root`, el cual representa la raíz del árbol. Este nodo es el punto de partida desde donde se extiende toda la estructura jerárquica.

El constructor de la clase inicializa este atributo en `null`, indicando que el árbol comienza vacío, sin nodos.

La funcionalidad principal que implementa esta clase es la inserción de nuevos elementos en el árbol. Para ello, cuenta con un método público llamado `insert`, que recibe como parámetro el valor entero que queremos agregar. Este método llama internamente a una función recursiva llamada `insertRecursive`, que es la que realmente se encarga de recorrer el árbol y colocar el nuevo nodo en la posición correcta.

El método `insertRecursive` recibe dos parámetros: el nodo actual que está evaluando y el valor que queremos insertar. Si el nodo actual es `null`, significa que se ha encontrado un lugar libre en el árbol, por lo tanto se crea y retorna un nuevo nodo con ese valor.

Si el valor a insertar es menor al valor del nodo actual, la función se llama a sí misma para continuar el recorrido por el subárbol izquierdo. En cambio, si el valor es mayor, se continúa por el subárbol derecho.

Este proceso se repite hasta encontrar una posición vacía donde pueda colocarse el nuevo valor, respetando las reglas del árbol binario de búsqueda: todos los elementos menores van a la izquierda y los mayores a la derecha.

Finalmente, se devuelve el nodo actual para ir actualizando los enlaces del árbol durante el proceso de inserción.

Esta clase es la encargada de mantener el árbol ordenado a medida que se agregan datos, lo que permitirá luego realizar búsquedas eficientes u otros recorridos de manera organizada.

Clase `BinarySearchTreeG.java`

Continuamos con la siguiente clase del programa, `BinarySearchTreeG`, que nos permite probar el funcionamiento completo del árbol binario de búsqueda. En este archivo se concentran todas las operaciones clave: insertar elementos, recorrer el árbol y verificar si contiene ciertos valores.

En primer lugar, se instancia un nuevo objeto de la clase `BinarySearchTree`. A través de este objeto es posible utilizar todos los métodos definidos previamente, como `insert`, `inOrderTraversal` y `contains`.

La inserción de elementos se realiza mediante el método `insert(int value)`, el cual funciona de manera idéntica a como se definió previamente. En esta sección, sin embargo, el valor se encapsula dentro del método para que pueda ser representado correctamente dentro del árbol, y así mostrarse visualmente cuando se ejecute el programa.

Luego se utiliza el método `inOrderTraversal()`, el cual activa un recorrido in-order del árbol. Este recorrido es importante porque nos permite visualizar los elementos de forma ordenada, desde el menor hasta el mayor. Internamente, este método llama a una función recursiva llamada `inOrderTraversalRecursive()`, que se encarga de recorrer el subárbol izquierdo, imprimir el nodo actual, y luego recorrer el subárbol derecho. Así, los valores se imprimen en orden ascendente. Esto tiene sentido porque, como ya insertamos los nodos siguiendo las reglas del ABB (árbol binario de búsqueda), los valores más pequeños se colocan en la izquierda y los mayores en la derecha.

En cuanto a la búsqueda, contamos con el método `contains(int value)`, el cual verifica si un valor específico se encuentra en el árbol. Esta función es especialmente útil para realizar consultas. Por ejemplo, en esta clase se evalúa si los valores 40 y 90 están presentes en el árbol. Aquí es donde el método `containsRecursive()` cobra importancia: recorre el árbol comparando cada nodo con el valor buscado y retorna `true` si lo encuentra o `false` si no está presente.

En el `main` del programa se insertan los siguientes valores: 50, 30, 70, 20, 40, 60 y 80. Esta lista representa una secuencia desordenada que será estructurada de manera jerárquica mediante el árbol. Posteriormente, se imprime el recorrido in-order de este árbol, el cual mostrará los valores en orden ascendente gracias al funcionamiento de `inOrderTraversal`.

Finalmente, el programa hace dos preguntas usando el método `contains`. Primero, pregunta si el número 40 está presente, lo cual es verdadero. Luego, consulta si el 90 forma parte del árbol, lo cual es falso, ya que ese número no fue insertado.

Esta clase, en resumen, actúa como un entorno de pruebas para validar el correcto funcionamiento del árbol binario. Gracias a ella podemos insertar nodos, recorrer el árbol visualmente y hacer búsquedas, todo dentro de un mismo archivo de ejecución.

Clase `TreePanel.java`

Finalmente, pasamos a la cuarta clase del proyecto, llamada `TreePanel`. Esta clase es responsable de la representación visual del árbol binario de búsqueda que hemos construido previamente.

Al inicio de la clase se declara el atributo `root`, que es una referencia al nodo raíz del árbol. Este nodo se recibe como parámetro en el constructor y permite que la clase sepa desde dónde comenzar a dibujar. También se definen constantes para el radio de los nodos y la separación vertical entre niveles del árbol.

El método clave de esta clase es `paintComponent`, el cual sobrescribe el método correspondiente de la clase `JPanel`. Este método es llamado automáticamente por el sistema gráfico de Java cada vez que el panel necesita redibujarse. Si la raíz del árbol no es nula, se llama al método auxiliar `drawTree`, encargado de realizar el dibujo nodo por nodo.

`drawTree` es un método recursivo que recibe el nodo actual, las coordenadas `x` e `y` para su ubicación, y un desplazamiento horizontal `xOffset`. Este método dibuja un círculo negro para representar el nodo, coloca el valor en blanco dentro del círculo, y luego dibuja las líneas hacia los hijos izquierdo y derecho si existen. Para cada hijo, se calculan nuevas posiciones en pantalla usando un desplazamiento relativo que disminuye conforme se desciende en el árbol. Esto asegura que los nodos no se encimen y que la estructura del árbol se mantenga visualmente clara.

Las variables `childX` y `childY` representan las coordenadas de los hijos. A medida que los nodos se ramifican hacia abajo, la separación horizontal entre ellos se reduce a la mitad, lo que imita la estructura jerárquica del árbol sin colisiones visuales.

En el método `main`, se crea una instancia de la clase `BinarySearchTreeG` y se insertan varios valores en el árbol. Posteriormente, se construye una ventana utilizando `JFrame` y se añade una instancia de `TreePanel`, pasando como argumento la raíz del árbol. Al ejecutar el programa, se abre una ventana gráfica donde se dibuja todo el árbol binario de búsqueda.

En resumen:

- `TreeNode` define la estructura de cada nodo del árbol.
- `BinarySearchTree` construye la lógica del árbol binario.
- `BinarySearchTreeG` permite insertar, recorrer y buscar elementos.
- `TreePanel` se encarga de visualizar gráficamente el árbol completo.

Cuando compilamos el programa, observamos que los nodos hijos más cercanos a la raíz tienen mayor separación horizontal. Esto es necesario para que la visualización del árbol sea clara y legible, especialmente en los niveles superiores donde hay más ramificación.

Programa PilaTree

Ahora pasamos a la parte correspondiente a **PilaTree**, en donde se introduce el recorrido *inorden* de un árbol binario utilizando una **pila**. Este recorrido nos permite visitar todos los nodos en un orden específico: primero los nodos del subárbol izquierdo, después la raíz, y por último los del subárbol derecho.

La clase encargada de este recorrido es `BinaryTreeTraversal`. Dentro de ella encontramos el método `recorridoInOrden`, que recibe como parámetro la raíz del árbol. Para realizar el recorrido sin utilizar recursividad, se hace uso de la estructura de datos `Stack`, es decir, una pila.

El proceso inicia declarando una pila de tipo `TreeNode`, y una variable `actual` que apunta inicialmente a la raíz del árbol. El ciclo `while` principal se mantiene activo mientras `actual` no sea `null` o la pila no esté vacía.

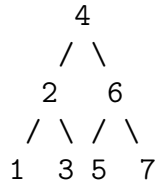
En la primera parte del recorrido, se avanza por el lado izquierdo del árbol: se van apilando los nodos recorridos con `pila.push(actual)`, y en cada paso se actualiza la variable `actual` al nodo hijo izquierdo (`actual = actual.izquierdo`). En esta parte, también se va mostrando el estado de la pila con el método `mostrarPila`, que imprime la acción realizada (por ejemplo, `Push 2`) junto con el contenido actual de la pila.

Una vez que se llega a un nodo sin hijo izquierdo, se empieza a desapilar. Se toma el nodo del tope con `pila.pop()` y se imprime que ha sido visitado. Esto indica que ya se terminó de recorrer su lado izquierdo, así que es momento de "visitar.[el] nodo (imprimir su valor) y luego moverse al hijo derecho.

El método `mostrarPila` es una función de apoyo que imprime el estado actual de la pila, mostrando cada valor que contiene junto con la acción que se está realizando, ya sea una inserción (`push`) o una extracción (`pop`).

Por su parte, la clase `TreeNode` define la estructura de cada nodo del árbol. Tiene un valor entero (`valor`) y dos referencias: una al hijo izquierdo y otra al hijo derecho. Esta clase es la base que permite construir la estructura jerárquica del árbol binario.

En el método `main` se crea manualmente un árbol binario completo de siete nodos con la siguiente estructura:



Después se llama al método `recorridoInOrden`, el cual imprime los valores del árbol en orden creciente: `1 2 3 4 5 6 7`.

Este enfoque permite recorrer el árbol sin utilizar recursividad, lo cual puede ser útil en situaciones donde se desea un control más detallado del recorrido, o se busca evitar el uso excesivo del stack de llamadas.

En resumen:

- `TreeNode` representa la estructura de los nodos del árbol binario.
- `BinaryTreeTraversal` implementa el recorrido inorden utilizando una pila.
- Se visualiza cómo se apilan y desapilan nodos mientras se avanza por el árbol.
- Se evita la recursividad usando una estructura de tipo LIFO.